Longan Linux

使用指南

版本号: 1.9

发布日期: 2025.3.17

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.06.24	AW1637	发布初版
1.1	2022.05.20	AW1538	修改排版
1.2	2022.05.20	AW1739	增加 bsp 独立仓库说明
1.3	2022.11.21	XA0193	根据最新的 longan 环境，对内容进行补充
1.4	2022.12.14	XA0193	根据转 PDF 的情况，将部分导出时格式会异常的表格换成普通按行分割的方式
1.5	2023.11.15	AW1727	校对内容，修改错别字、语法错误等
1.6	2024.09.30	AW1730	新增 T536 平台支持; 根据转 PDF 的情况，将部分格式异常进行修复
1.7	2024.12.11	AW2099	新增 A733 平台支持; 新增对 buildroot 适用范围的描述
1.8	2024.12.11	XAA0193	新增 A537 A333 平台支持;
1.9	2025.3.17	XAA0329	1. 更正 shell 注释 2. 更正有歧义的语句 3. 更正表格的命令与功能不匹配问题 4. 更正秘钥生成路径 5. 更正 S 脚本优先级介绍

目　录

插　图

1 前言

1.1 编写目的

本文档介绍全志科技 longan 系统 (Linux/Android SDK) 的开发环境、目录结构、编译和打包，主要目的用于指导用户如何定制和使用 Linux/Android SDK。

1.2 适用范围

基于 longan 系统的 Linux SDK 开发平台，目前支持的平台有 A733，A537、A333、T536，T527，A523，A133，A50，A63，A64，H3，H6，R328，R328S2，R328S3，R329，T3，T507，T7，T8，V316，V459，V5，V536 等。

1.3 相关文档

此份文档重点介绍整个 longan 系统的使用办法，至于内容中所提到的 BSP 仓库说明请见同级目录下的《Linux_BSP 独立仓库 _ 开发指南》。

1.4 适用人员

基于 longan 构建环境的 Linux/Android SDK 开发者。

2 开发环境

2.1 系统介绍

2.1.1 longan 系统介绍

longan 是 lichee 和 kunos 合并后的名称，同时也兼容新的 Tina5.0 方案。它集成了 BSP，构建系统，独立 IP 和测试，既可作为 BSP 开发和 IP 验证平台，也可以作为量产的嵌入式 linux 系统。

longan 的功能包括以下四部分：

1. BSP 开发，包括 uboot 和 kernel。
2. Linux SDK 开发，包括量产的嵌入式 linux 系统。
3. IP 的验证和发布平台，包括 gpu，cedarx，gstreamer，drm/weston，security 以及其他的私有软件包。IP 随 longan 的发布而发布，减少使用邮件发布；并且给出 IP 的使用方法和系统集成的 demo 程序，方便第三方快速使用。
4. 测试，包括板级测试和系统测试，如 drangonboard。

2.1.2 buildroot 介绍

buildroot，是嵌入式开发领域内一个成套的嵌入式开发环境。它可以用来定制自己的交叉编译器，制作自己的根文件系统，能把 uboot，linux kernel 集成到它的编译框架中一起编译。

buildroot 集成在 longan 系统里面，可以在 longan 里面配置并且编译。

特别注明：纯 Android SDK 的 longan 是不包含 buildroot 的，对于纯 Android SDK 请忽略下文中的 buildroot 相关描述。

2.2 软件环境

- Ubuntu-12.04 及以上版本。
- 编译环境依赖于 libelf-dev 库，安装命令如下所示：

```
apt install libelf-dev
```

- 当 libelf-dev 库缺少时的异常打印如下所示：

```
btf.c:18:10: fatal error: 'gelf.h' file not found
#include <gelf.h>
         ^~~~~~~
 CC     /disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids/libbpf/staticobjs/
        btf_dump.o
 CC     /disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids/libbpf/staticobjs/ringbuf.
        o
 CC     /disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids/libbpf/staticobjs/strset.o
 CC     /disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids/libbpf/staticobjs/linker.o
1 error generated.
make[5]: *** [/disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids/libbpf/staticobjs/
        libbpf.o] Error 1
make[5]: *** Waiting for unfinished jobs....
make[3]: *** [/disk4/jfzhen/A133_T13.0_MID/longan/out/a133/kernel/build/tools/bpf/resolve_btfids//resolve_btfids-in.
        o] Error 2
make[3]: *** Waiting for unfinished jobs....
linker.c:17:10: fatal error: 'libelf.h' file not found
#include <libelf.h>
         ^~~~~~~~~~
1 error generated.
```

2.3 目录结构

```
longan/
├── brandy                  //uboot和cpu的源码
│   ├── arisc               //cpus源码目录
│   └── brandy-2.0          //uboot源码目录
├── bsp                     //BSP独立仓库源码目录
│   ├── configs             //配置文件目录
│   ├── drivers             //AW的私有驱动源码目录
│   ├── include             //AW的私有头文件目录
│   ├── Kconfig
│   ├── Makefile
│   ├── modules             //AW的模块源码目录
│   ├── platform            //AW的架构配置目录
│   └── ramfs               //AW的ramfs文件系统目录
├── build                   //编译打包脚本
│   ├── bin                 //制作文件系统工具
│   ├── bsp.sh              //与BSP独立仓库联合编译的脚本
│   ├── buildbase.sh
│   ├── createkeys          //创建安全方案秘钥工具
│   ├── disclaimer
│   ├── envsetup.sh         //配置环境变量
│   ├── getvmlinux.sh
│   ├── hook
│   ├── Makefile
│   ├── mkcmd.sh            //主要编译脚本
│   ├── mkcommon.sh         //编译脚本
│   ├── mkkernel.sh         //内核编译相关的脚本
│   ├── pack                //打包脚本
│   ├── parser.sh
│   ├── quick.sh
│   ├── shflags
│   └── swapsdc.sh
```

```
├── swupdate
│   └── top_build.sh
├── build.sh -> build/top_build.sh      //顶层编译脚本
├── documentation                       //各个平台的文档说明
│   ├── config
│   ├── intermediates
│   ├── output
│   ├── source
│   └── tools
├── device                              //存储方案配置的目录
│   ├── config
│   │ │chips                          //板级配置
│   │ │ ├── A133                      //A133配置
│   │ │ │   ├── bin                   //uboot和烧写程序
│   │ │ │   ├── boot-resource         //启动资源文件，如bootlogo图片
│   │ │ │   ├── config                //方案板配置
│   │ │ │   │   ├── default           //默认配置和通用配置
│   │ │ │   │   ├── p25               //A133-p25方案板配置
│   │ │ │   │ │   ├── android         //andriod sdk配置
│   │ └── product -> ./config/chips/a133/
├── kernel
│   └── linux-5.15                    //内核原生代码目录
├── out
│   ├── a133                          //a133相关的编译生成物
│   ├── a133_linux_p25_card0_secure_v0.img //固件
│   ├── kernel -> a133/kernel
│   ├── pack_out -> a133/p25/pack_out
│   ├── serversocket
│   └── toolchain                     //解压后的编译工具链
├── platform                          //私有软件包定制目录，使用buildroot系统编译
│   ├── apps                          //app代码
│   ├── base                          //基础库文件
│   ├── config                        //allwinner私有软件包配置
│   │   ├── buildroot                 //针对buildroot的私有软件包配置
│   ├── core                          //服务，组件
│   ├── external                      //第三方库、组件、服务
│   ├── framework                     //中间层框架
│   ├── Makefile -> ../../build/Makefile
│   └── tools                         //调试代码、工具
├── prebuilt                          //存放各种编译环境库
│   ├── hostbuilt                     //pc工具
│   └── kernelbuilt                   //内核编译工具链源码
├── tee_kit
│   ├── build.sh -> dev_kit/build.sh
│   ├── demo
│   ├── dev_kit
│   └── README.md
├── test                              //AW的测试文件
│   ├── auto_testplains
│   ├── bsptest                       //bsptest测试系统
│   ├── dragonboard                   //dragonboard测试系统
└── tools                             //pc工具
    ├── build
    ├── codecheck                     //代码检查工具
    ├── pack
    └── tools_win                     //windows软件工具
```

2.4 longan 编译方式

2.4.1 SDK 常用的编译命令和编译步骤介绍

longan 生成 image 的步骤包括配置，编译，打包三个步骤。这三个步骤都在 longan 的根目录下进行，具体的步骤如下所示：

步骤 1，进行 sdk 环境配置，详细看下一节

```
./build.sh config
```

步骤 2，编译整个 SDK

```
./build.sh
```

步骤 3，打包固件

```
./build.sh pack              #打包固件，android方案无需此打包
./build.sh pack_debug        #打包卡打印固件，android方案无需此打包
./build.sh pack_debug_secure  #打包安全卡打印固件，android方案无需此打包
./build.sh pack_secure       #打包安全固件，android方案无需此打包
```

注意如果需要打包成安全固件，需要先生成安全密钥，跳转到 build 目录，执行 createkeys 文件，然后选择相应的平台。

2.4.2 longan 编译配置示例

下面就以编译 T527 平台 demo 板子 longan 系统为例展示一个 longan 配置过程，跳转到 longan 根目录：

```
longan$ ./build.sh config
========ACTION List: mk_config ;========
options :
All available platform:
  0. android
  1. linux
Choice [linux]:1
All available linux_dev:
  0. bsp
  1. dragonboard
  2. buildroot
Choice [bsp]:0
All available kern_ver:
  0. linux-5.10
  1. linux-5.15
Choice [linux-5.15]:1
All available ic:
  0. a523
  1. a527
  2. t527
Choice [t527]:2
All available board:
```

```
  0. demo
  1. demo_car
  2. demo_fastboot
  3. demo_linux_aiot
  4. demo_linux_car
Choice [demo_car]:1
All available flash:
  0. default
  1. nor
Choice [default]:0
```

2.4.3 buildroot 配置和编译

可以对 buildroot 进行单独配置和编译，在 buildroot/buildroot-xxx 目录下执行。

1. 界面配置和编译

界面配置命令如下

```
make sun8iw18p1_longan_defconfig   //从configs/sun8iw18p1_longan_defconfig生成到 out/sun8iw18p1/std/longan/
      buildroot/.config
make menuconfig              //配置界面
make savedefconfig           //保存配置到configs/sun8iw18p1_longan_defconfig
```

注意事项如下所示：

- 不要手动 copy 生成 config，否则上面的命令无效
- buildroot/buildroot-xxx/.config 是无效的，longan 系统编译时不会生成或者使用到此文件。

编译命令如下

- 方法 1：buildroot/buildroot-xxx 目录下执行：make
- 方法 2：longan 顶层目录执行：./build.sh buildroot

2.4.4 longan 的常用命令

longan 常见编译打包命令如下所示：

	命令	用法说明
编译配置	./build.sh config	1. 编译配置，然后弹出配置选择 2. 作用是找到对应的 BoardConfig.mk 文件，并且配置部分参数
整体编译	./build.sh	编译命令，编译 kernel、buildroot
局部编译	./build.sh bootloader	编译 boot0、uboot、efex

	命令	用法说明
局部编译	./build.sh kernel	编译 kernel
内核 defconfig 配置	./build.sh menuconfig	打开内核的配置界面
保存内核 defconfig	./build.sh saveconfig	保存内核配置（menuconfig）
局部编译	./build.sh buildroot	编译 buildroot
局部编译	./build.sh buildroot_menuconfig	打开 buildroot 的配置界面
局部编译	./build.sh buildroot_saveconfig	保存 buildroot 配置（menuconfig）
打包	./build.sh pack	打包命令，生成 uart0 固件
打包	./build.sh pack_debug	打包命令，生成 card 固件
打包	./build.sh pack_debug_secure	打包安全卡打印固件

2.4.5 buildroot 常用的编译命令

	命令	用法说明
整体编译	make xxx_defconfig	从 Configs/xxx_defconfig 生成 out/{IC}/{board}/longan/buildroot/.config
整体编译	make menuconfig	界面配置
整体编译	make savedefconfig	保存 out/{IC}/{board}/longan/buildroot/.config 到 Configs/xxx_defconfig
整体编译	make	整体编译
整体编译	make clean	清除过程文件和目标文件
整体编译	make distclean	清除所有的生成文件
软件包	make xxx-build	编译名为 xxx 的软件包
软件包	make xxx-dirclean	清除名为 xxx 的软件包
软件包	make xxx-rebuild	重新编译名为 xxx 的软件包
软件包	make xxx-reconfigure	重新配置 xxx 软件包

其中，IC 指的是芯片，如 A100，R328 等等，board 指的是板级，如 perf1，std，ver1 等。

3 编译系统说明

3.1 longan 编译系统

3.1.1 longon 编译系统介绍

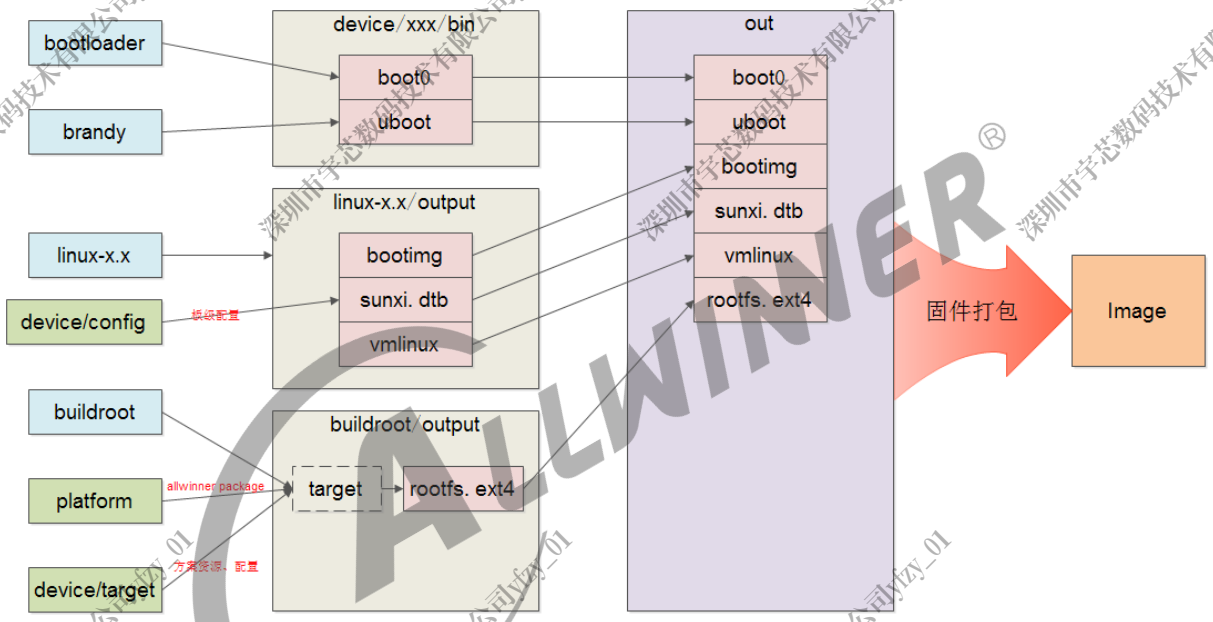

图 3-1: longan 编译打包示意图

Longan 编译打包示意图如上所示，Platform 为全志私有源码包，它独立于 buildroot 之外，但是和 buildroot 一起联合编译。其目的是为了，buildroot 快速升级和适配第三方 buildroot。

如图中所示，longan 有三部分 boot、kernel、buildroot，三部分相互独立，因此有三个交叉编译工具链，boot 工具链不运行更改，kernel 和 buildroot 可以更改，方法见后面章节《更换交叉编译工具链》。

Longan 编译由 BoardConfig.mk 决定编译规则，详细见方法见下一节《修改编译规则 BoardConfig.mk》。

各部分的目标输出如下所示：(以 buildroot 举例)

- device/config/chips/{IC}/bin：通用 boot，默认生成在芯片配置目录的 bin 目录中

- device/config/chips/{IC}}/configs/{board}/longan/bin：定制化 boot，存放在板级方案的系统目录下 bin 目录
- out/{IC}/{board}/kernel/build/：内核编译的目标文件存放目录
- out/{IC}/{board}/kernel/build/.config：生成的内核配置文件
- out/{IC}/{board}/buildroot/buildroot/.config：生成的 buildroot 配置文件
- out/{IC}/{board}/buildroot/buildroot：目标文件，buildroot 编译的目标文件存放目录

3.1.2 编译配置规则

当./build.sh config 完成之后，编译系统会生成一个方案的编译规则，其保存在在 longan 根目录中的.buildconfig 文件里面。下面是一个规则的说明。

- LICHEE_LINUX_DEV：编译的方案，在./build.sh config 第二个选项中指定，如 bsp
- LICHEE_IC：编译的 IC，在./build.sh config 第四个选项中指定，如 r328
- LICHEE_BOARD：编译的板级配置，在./build.sh config 第五个选项中指定，如 perf1
- LICHEE_PLATFORM：编译平台，在./build.sh config 第一个选项中指定，如 linux
- LICHEE_FLASH：编译的 flash 配置，在./build.sh config 第六个选项中指定，如 deflaut
- LICHEE_CHIP：编译的芯片代号名称，在 BoardConfig.mk 指定
- LICHEE_KERN_VER：编译的内核版本，在./build.sh config 第三个选项中指定，如 linux-4.9
- LICHEE_KERN_DEFCONF：编译内核的默认配置，在 BoardConfig.mk 指定
- LICHEE_BUILDING_SYSTEM：编译的构造系统，在 BoardConfig.mk 指定
- LICHEE_BR_VER：buildroot 的版本，在 BoardConfig.mk 指定
- LICHEE_BR_DEFCONF：buildroot 的默认配置，在 BoardConfig.mk 指定
- LICHEE_BR_RAMFS_CONF：buildroot 的 ramfs 默认配置，在 BoardConfig.mk 指定
- LICHEE_BRANDY_VER：uboot 的版本，在 BoardConfig.mk 指定
- LICHEE_BRANDY_DEFCONF：brandy 的默认配置，在 BoardConfig.mk 指定
- LICHEE_CROSS_COMPILER：编译使用的交叉编译链，在/longan/build/mkcmd.sh 中指定，也可以在 BoardConfig.mk 指定
- LICHEE_CHIP_CONFIG_DIR：编译系统的 IC 配置目录，由编译系统自动生成
- LICHEE_OUT_DIR：编译的输出目录，有编译系统自动生成

上面只列出了部分重要的配置，详细的配置可以查看该文件。

而某个方案的大部分编译规则都是由 BoardConfig.mk，其存放在每个方案的板级配置目录中，同一个板型的不同系统拥有不用的 BoardConfig.mk 配置。

3.2 buildroot 开发和编译

3.2.1 与标准 buildroot 的不同

由于 longan 需要根据不同板型来配置并编译 buildroot，这就决定了 Longan 的 buildroot 和标准 buildroot 编译有些不同。

Longan 系统中的 buildroot 和标准 buildroot 的不同主要在.config 和编译目标输出。位置和说明情况如下所示：

- buildroot/buildroot-xxx/.config：longan 系统编译时不会生成或者使用到此文件，属于无效文件
- out/{IC}/{board}/buildroot/buildroot/.config：buildroot 编译时使用到的.config
- out/{IC}/{board}/buildroot/ramfs/.config：ramfs 编译时使用到的.config
- buildroot/buildroot-xxx/output：longan 系统编译时不会生成或者使用到此目录，属于无效目录
- out/{IC}/{board}/buildroot/buildroot/：同一个芯片的不同板型有不同的 buildroot 输出目录
- out/{IC}/{board}/buildroot/ramfs/：ramfs 输出目录

3.2.2 buildroot 目录结构

buildroot 目录结构如下所示：

```
.
├── arch  //存放CPU架构相关的配置脚本，如arm/mips/x86
├── board  //存放了一些默认开发板的配置补丁之类的
├── boot    //引导系统
├── build.sh
├── CHANGES
├── Config.in
├── Config.in.legacy
├── configs  //放置开发板的一些配置参数,如sun8iw*_*_defconfig
├── COPYING
├── DEVELOPERS
├── dl   //存放下载的源码包及应用软件的压缩包
├── docs //存放相关的参考文档
├── fs //放各种文件系统的源代码
├── linux //存放着Linux kernel的自动构建脚本
├── Makefile
├── Makefile.legacy
├── package //下面放着应用软件的配置文件，每个应用软件的配置文件有Config.in和soft_name.mk，其中soft_name.
    mk(这种其实就Makefile脚本的自动构建脚本)文件可以去下载应用软件的包
├── README
├── scripts //存放编译脚本
├── support
├── system //这里就是根目录的主要骨架了和相关的启动初始化配置,当制作根目录时就是将此处的文件cp到output里去.
    然后再安装toolchain的动态库和你勾选的package的可执行文件之类的
│   └── skeleton // 放生成文件系统镜像的地方，及板子里面的系统
└── toolchain //存放编译工具链
```

```
├── output // 输出文件夹，存放解压后的代码和目标文件
│   ├── build //存放解压后的各种软件包和编译完成后的现场
│   ├── host //存放着制作好的编译工具链，如gcc、arm-linux-gcc等工具
│   ├── images //存放着编译好的uboot.bin, zImage, rootfs等镜像文件，可烧写到板子里, 让linux系统跑起来。
│   ├── staging //软链接到host/< tuple >/sysroot/ 就是上面说到的文件系统需要的so库、*.h等目录,方便查看
│   ├── target //用来制作rootfs文件系统，里面放着Linux系统基本的目录结构，以及编译好的应用库和bin可执行文件. (
    buildroot根据用户配置把.ko .so .bin文件安装到对应的目录下去，根据用户的配置安装指定位置)
└── utils
```

注：buildroot 根目录下的 output 目录，在编译系统里面，转移到了 out/{IC}/{board}/buildroot/buildroot 下。

3.2.3 编译修改 config 配置

3.2.3.1 buildroot 配置修改

以 R328 为例，修改步骤如下：

1. 编译得到.config: make sun8iw18p1_longan_defconfig
2. 界面化配置：make menuconfig，配置完成后退出
3. 保存方案配置：make savedefconfig

3.2.3.2 busybox 配置修改

1. 配置命令：make busybox-menuconfig
2. 修改完成后，保存方案配置：make busybox-update-config

3.2.4 单个软件包配置

3.2.4.1 编译单个软件包

单个软件包编译在 buildroot 目录下进行，其方法：make package-build

例如 android-tools 编译：

```
android-tools位于buildroot/package/android-tools
make android-tools-build
```

3.2.4.2 删除单个软件包

删除单个软件包的命令：make package-dirclean

在没有修改 config 的情况下，不支持 Removing a package, 是因为 buildroot 没有记录在 output 中安装的相应信息, 和依赖的包. 但只要删了 output/build 下的相应目录, 再 make 时它就会重新解压配置编译了, 所以可以使用如下命令来删除相应目录。

3.2.4.3 重新编译一个软件包

重新编译一个软件包的命令如下所示：make package-rebuild

3.2.4.4 查询要使用的源码包

列出所有要用的源码包：make external-deps

4 longan 快速支持平台

4.1 新增板级方案

本节主要介绍如何快速支持一颗新的 IC，并能快速支持 bsp buildroot openwrt android 等方案，主要遵循以下步骤，便可快速在 longan 环境下支持新的平台。下面以 r328 为例（下面所有路径都以 longan 目录为相对路径）

4.1.1 创建板级目录

在 device/config/chips 目录下添加一个以 IC 名称命名的目录，如 r328，目录主要存放以下内容：

```
├── bin   //存放的bin文件，相关同事提供
│   ├── boot0_nand_sun8iw18p1.bin
│   ├── boot0_sdcard_sun8iw18p1.bin
│   ├── boot0_spinor_sun8iw18p1.bin
│   ├── fes1_sun8iw18p1.bin
│   ├── optee_sun8iw18p1.bin
│   ├── sboot_sun8iw18p1.bin
│   ├── u-boot-spinor-sun8iw18p1.bin
│   └── u-boot-sun8iw18p1.bin
├── boot-resource   //存放开机图片，从其他平台copy，按需改动
│   ├── boot-resource
│   │   ├── bat
│   │   │   ├── bat0.bmp
│   │   │   ├── bat10.bmp
│   │   │   ├── bat1.bmp
│   │   │   ├── bat2.bmp
│   │   │   ├── bat3.bmp
│   │   │   ├── bat4.bmp
│   │   │   ├── bat5.bmp
│   │   │   ├── bat6.bmp
│   │   │   ├── bat7.bmp
│   │   │   ├── bat8.bmp
│   │   │   ├── bat9.bmp
│   │   │   ├── battery.bmp
│   │   │   ├── battery_charge.bmp
│   │   │   ├── bempty.bmp
│   │   │   ├── bootlogo.bmp
│   │   │   └── low_pwr.bmp
│   │   ├── bootlogo.bmp
│   │   ├── fastbootlogo.bmp
│   │   ├── font24.sft
│   │   ├── font32.sft
│   │   └── warnings.bmp
│   └── boot-resource.ini
├── configs   //存放打包，配置文件
```

```
├── default
│   ├── BoardConfig.mk   //默认配置文件
│   ├── BoardConfig_nor.mk //nor配置文件
│   ├── boot_package.cfg //从其他平台copy，按需改动
│   ├── boot_package_nor.cfg //从其他平台copy，按需改动
│   ├── diskfs.fex  //从其他平台copy，按需改动
│   ├── dragon_toc_android.cfg
│   ├── dragon_toc.cfg
│   ├── env_burn.cfg
│   ├── env.cfg  //uboot使用的环境参数
│   ├── env_nor.cfg
│   ├── image.cfg //从其他平台copy，按需改动
│   ├── image_nor.cfg
│   ├── sys_partition.fex //从其他平台copy，按需改动
│   ├── sys_partition_nor.fex
│   └── version_base.mk
├── demo  //具体板级方案
│   ├── board.dts
│   └── longan   //longan系统使用的配置
│       ├── BoardConfig.mk  //根据方案需要配置
│       ├── BoardConfig_nor.mk //根据方案需要配置
│       ├── bootlogo.fex
│       ├── env.cfg  //根据方案需要配置
│       └── sys_partition.fex //longan系统使用的分区表
│   └── sys_config.fex //系统配置文件
├── fpga
│   ├── board.dts
│   └── sys_config.fex
├── ft
│   ├── board.dts
│   └── sys_config.fex
├── perf1
│   ├── board.dts
│   ├── longan
│   │   ├── BoardConfig.mk
│   │   ├── BoardConfig_nor.mk
│   │   ├── bootlogo.fex
│   │   ├── env.cfg
│   │   └── sys_partition.fex
│   │   ├── sata
│   │   │   └── sys_partition_linux.fex
│   │   └── sys_config.fex
│   └── std
│       ├── board.dts
│       ├── longan
│       │   ├── BoardConfig.mk
│       │   ├── BoardConfig_nor.mk
│       │   ├── bootlogo.fex
│       │   ├── env.cfg
│       │   └── sys_partition.fex
│       └── sys_config.fex
└── tools
    ├── cardscript.fex
    └── readme.txt
```

从上述目录来看，bsp 开发人员需要特别关注 configs 目录下的文件则可。

4.1.2 修改编译规则 BoardConfig.mk

修改 BoardConfig.mk 可以达到修改编译规则的目的，下面是一个具体的示例：

```
LICHEE_ARCH:=arm                                    //芯片架构 arm or arm64
LICHEE_PRODUCT:=drum                                //芯片产品代号
LICHEE_BRANDY_VER:=2.0                              //brandy版本
LICHEE_KERN_VER:=4.9                                //kernel版本
LICHEE_KERN_DEFCONF:=sun8iw18p1smp_kunos_drum_defconfig          //kernel配置文件名称
LICHEE_BUILDING_SYSTEM:=buildroot                   //构建系统类型，buildroot or yocto
LICHEE_BR_VER:=201611                               //buildroot版本
LICHEE_BR_DEFCONF:=sun8iw18p1_longan_defconfig      //buildroot配置文件名称
LICHEE_BR_RAMFS_CONF:=sun8iw18p1_ramfs_defconfig    //buildroot ramfs config名称
LICHEE_COMPILER_TAR:=gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz  //kernel编译工具链
```

注意，如果是 linux 平台，则会使用板级包目录下的 BoardConfig.mk；如果是 android 平台，则会使用 android 目录下的 BoardConfig.mk。

4.1.3 修改差异化编译规则 BoardConfig-condations.mk

如果你想同时支持多个平台的编译，而这些平台间大部分的配置是一样的，此时你可以选择在将差异化的内容修改在 default 目录下 BoardConfig-condations.mk 文件中，而将相同的内容放在 default 或者具体版型下的 BoardConfig.mk 文件中。

下面是一个具体的例子：

```
# condition config

# arch

# kernel
ifeq ($(filter-out %5.4,$(LICHEE_KERN_VER)),)
  LICHEE_BRANDY_DEFCONF     := sun50iw10p1_android11_defconfig
  ifeq ($(LICHEE_PLATFORM),android)
    ANDROID_CLANG_PATH    := prebuilts/clang/host/linux-x86/clang-r416183b1/bin
  endif
else ifeq ($(filter-out %5.10 %5.15,$(LICHEE_KERN_VER)),)
  LICHEE_USE_INDEPENDENT_BSP := true
  LICHEE_BRANDY_DEFCONF     := sun50iw10p1_android11_defconfig
  ifeq ($(LICHEE_PLATFORM),linux)
    ifneq ($(LICHEE_LINUX_DEV),dragonboard)
      LICHEE_KERN_DEFCONF:= bsp_defconfig
    endif
  endif
endif

# platform
ifeq ($(LICHEE_PLATFORM),android)
  LICHEE_PACK_HOOK := build/hook/pack/hook.sh
endif
```

4.1.4 修改系统配置文件 sys_config.fex

sys_config.fex 用于配置系统参数，可以按照具体的平台来配置，具体的用法如下：

```
[name]
param = description
```

下面是一个示例：

```
[product]
version = "100"
machine = "perf1"
```

其定义了一个 product 节点，里面有 version 和 machine 参数，其会覆盖设备树中相关的 product 节点，如若没有，则会添加。

4.1.5 修改分区表 sys_partition.fex

如上面分区表 sys_partition.fex, 其定义了每个分区的名称和大小等等，具体如下所示：

```
; partition 定义范例:
;  [partition]          ; //表示是一个分区
;  name      = USERFS2    ; //分区名称
;  size      = 16384      ; //分区大小 单位: 扇区.分区表示个数最多2^32 * 512(Bytes) = 2T
;  downloadfile = "123.fex"  ; //下载文件的路径和名称，可以使用相对路径，相对是指相对于image.cfg文件所在分区。也可
       以使用绝对路径
;  keydata    = 1        ; //私有数据分区，重新量产数据将不丢失
;  encrypt    = 1        ; //采用加密方式烧录，将提供数据加密，但损失烧录速度
;  user_type  = ?        ; //私有用法
;  verify     = 1        ; //要求量产完成后校验是否正确
;
; 注：1、name唯一, 不允许同名
;    2、name最大12个字符
;    3、size = 0, 将创建一个无大小的空分区
;    4、为了安全和效率考虑，分区大小最好保证为16M字节的整数倍
```

示例如下：

```
;----------------------------->mmcblk0p4/nand0p4
[partition]
   name       = boot
   size       = 65536
   downloadfile = "boot.fex"
   user_type    = 0x8000
```

4.1.6 修改 bootlogo

如上面 bootlogo.fex，用来表示开机 logo，可以修改该文件来修改开机 logo

4.1.7 修改开机启动，资源文件等等

Base-files 存放需要添加到 rootfs 的是开机启动、配置、资源文件、第三方库或者可执行文件，其位于 longan 目录下的 device 目录，如下：

- device/target/common/base-files：存放通用配置
- device//target/drum/std/base-files：存放方案私有配置

增加新配置进 rootfs 的步骤：

1. 将需要预拷贝文件复制到 device/common/base-files 目录下
2. 修改 copy_files.mk 文件，增加和删减预拷贝文件。

4.1.8 修改开机动画和开机音乐

开机动画和开机音乐位于：device//target/drum/{board}/boot-play 下，可以新建一个目录用以存放

注意的是，上面所有的配置文件都是目录深度越大的优先级越高，如果当前板级目录下没有，则会采用 default 目录下的配置。如下面的第三个优先级最高。

1. LICHEE_CHIP_CONFIG_DIR/config/default 目录
2. LICHEE_CHIP_CONFIG_DIR/config/LICHEE_BOARD 目录
3. LICHEE_CHIP_CONFIG_DIR/config/LICHEE_BOARD/LICHEE_LINUX_DEV 目录

注：如果./build.sh config 获得的参数与 BoardConfig.mk 不同，则优先使用./build.sh config 的参数。

4.2 新增第三方 buildroot 支持

4.2.1 前提条件

第三方的 buildroot 编译和架构满足标准 buildroot 的要求

4.2.2 加入到 longan 目录

buildroot-201902 加入 r328-perf1 方案为例：

- 步骤 1：拷贝源码，将 buildroot-201902 拷贝到 buildroot 目录下。
- 步骤 2：修改 device/config/chips/r328/configs/perf1/longan/BoardConfig.mk 文件，新增如下配置：

```
LICHEE_BR_VER:=201902              //buildroot版本
LICHEE_BR_DEFCONF:=xxx_defconfig     //buildroot config配置文件
```

- 步骤 3：将 buildroot-201611/package/allwinner 拷贝到 buildroot-201902/package/
- 步骤 4：修改 buildroot-201902/package/Config.in 文件，如下：

```
menu "Target packages"

  source "package/busybox/Config.in"
  source "package/skeleton/Config.in"
  source "package/skeleton-custom/Config.in"
  source "package/skeleton-init-common/Config.in"
  source "package/skeleton-init-none/Config.in"
  source "package/skeleton-init-systemd/Config.in"
  source "package/skeleton-init-sysv/Config.in"
  source "../../platform/config/buildroot/Config.in"
```

4.3 buildroot 新增私有源码包

4.3.1 在 buildroot 目录中增加私有源码包

请参考《The Buildroot User Manual.pdf》。

4.3.2 在 platform 目录中增加私有源码包

以在 r328 中 boot-play（开机动画程序）程序为例，boot-play 程序涉及的目录和文件和说明情况如下所示：

- 目录：buildroot-201611，文件：configs/sun8iw18p1_longan_defconfig，说明：编译用的配置文件
- 目录：device/target，文件：conmmon/base_files/etc/init.d/S39initplay，说明：开机启动脚本（见后面章节《添加开机启动脚本和外部资源文件》）

- 目录：device/target，文件：conmmon/base_files/boot-play，说明：存放开机动画、开机音乐等资源文件
- 目录：platform/config，文件：buildroot/boot-play/Config.in，说明：在 buildroot 目录中使用命令 make menuconfig 参与构建
- 目录：platform/config，文件：buildroot/boot-play/boot-play.mk，说明：buildroot 编译时使用，记录了软件包版本、位置、依赖关系、编译规则等。
- 目录：platform/config，文件：buildroot/boot-play/S39initplay，说明：开机启动脚本
- 目录：platform/config，文件：buildroot/post_build.sh，说明：（见后面章节《添加开机启动脚本和外部资源文件》）
- 目录：platform/apps，文件：boot-play，说明：开机动画的源代码和 makefile

4.3.3 源码包编译需要满足的条件

Buildroot 支持如下编译方式：generic-package、cmake-package、autotools-package。源码包的编译规则需要上面三种方式的一种。

4.3.4 新增源码方法步骤

4.3.4.1 新增源码包

根据代码在系统中的层次关系，在对应的目录新建代码目录，请参考 longan 的《目录结构》。以 boot-play 程序为例，boot-play 源码包位于：platform/apps/boot-play。

- 步骤 1：创建 platform/apps/boot-play 目录，将源码拷贝到 platform/apps/boot-play 中。
- 步骤 2：编写 platform/apps/boot-play/Makefile

```
SRCS = $(wildcard *.c)              //源码*.c
OBJS = $(patsubst %.c, %.o, $(SRCS))
DEPS = $(SRCS:.c=.dep)
OUT_BIN = initplay                  //编译输出的目标名

all: $(OUT_BIN)

#ifndef ($(MAKECMDGOALS), clean)
-include $(DEPS)
#endif

$(OUT_BIN): $(OBJS)
   $(CC) $(LDFLAGS) -o $@  $(filter %.o, $^)

%.o: %.c
   $(CC) $(CFLAGS) -o $@ -c $(filter %.c, $^)

%.dep: %.c
   @echo "Creating $@ ..."
```

```
	@set -e; \
	rm -rf $@.tmp; \
	$(CC) -E -MM $(filter %.c, $^) > $@.tmp; \
	sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@; \
	rm -rf $@.tmp

.PHONY: clean
clean:
	rm -rf *.o
	rm -rf $(OUT_PUT)
	rm -rf *.dep
```

4.3.4.2 编写 buildroot 编译配置脚本

- 步骤 1：新建 platform/config/buildroot、boot-play 目录用于存放 boot-play 编译配置。
- 步骤 2：新建 platform/config/buildroot/boot-play/Config.in 文件，编写如下

```
config BR2_PACKAGE_BOOT_PLAY  //建议：BOOT_PLAY和软件包boot-play同名
	bool "boot-play package"
	select BR2_PACKAGE_LIBUBOX
	help
	  boot-play package
```

- 步骤 3：新建 platform/config/buildroot/boot-play/boot-play.mk（建议：boot-play 和软件包 boot-play 同名）文件，编写如下：

```
################################################################################
#
# boot-play package
#
################################################################################
BOOT_PLAY_SITE_METHOD = local
BOOT_PLAY_SITE = $(PLATFORM_PATH)/../../apps/boot-play
BOOT_PLAY_LICENSE = GPLv2+, GPLv3+
BOOT_PLAY_LICENSE_FILES = Copyright COPYING
BOOT_PLAY_DEPENDENCIES = libubox libpng
BOOT_PLAY_CFLAGS = $(TARGET_CFLAGS)
BOOT_PLAY_CFLAGS += -O2 -fPIC -Wall -DCOMPLETE_TIMEOUT
BOOT_PLAY_CFLAGS += -I$(STAGING_DIR)/usr/include -I$(@D)
BOOT_PLAY_LDFLAGS = $(TARGET_LDFLAGS)
BOOT_PLAY_LDFLAGS += -L$(TARGET_DIR)/usr/lib/ -lpng -lubox

define BOOT_PLAY_BUILD_CMDS
	$(MAKE) CC="$(TARGET_CC)" CFLAGS="$(BOOT_PLAY_CFLAGS)" \
	  LDFLAGS="$(BOOT_PLAY_LDFLAGS)" -C $(@D) all
endef

define BOOT_PLAY_INSTALL_TARGET_CMDS
	$(INSTALL) -D -m 0755 $(@D)/initplay $(TARGET_DIR)/usr/bin
$(INSTALL) -D -m 0755 $(PLATFORM_PATH)/boot-play/S39initplay $(TARGET_DIR)/etc/init.d
//拷贝开机启动脚本到etc目录
endef

$(eval $(generic-package))
```

参考《The Buildroot user manual》第 15 章《15.2 The .mk file》

- 步骤 4：增加 Config.in 到整体编译中，修改 platform/config/buildroot/Config.in，如下：

```
.....
source "../platform/config/buildroot/demo/Config.in"
source "../platform/config/buildroot/libcedarc/Config.in"
source "../platform/config/buildroot/cedarx/Config.in"
source "../platform/config/buildroot/boot-play/Config.in"
source "../platform/config/buildroot/mediaplayer/Config.in"
......
```

- 步骤 5： 增加 boot-play.mk 到整体编译中，修改 platform/config/buildroot/platform.mk，如下：

```
.....
include ${PLATFORM_PATH}/cedarx/cedarx.mk
include ${PLATFORM_PATH}/boot-play/boot-play.mk
include ${PLATFORM_PATH}/mediaplayer/mediaplayer.mk
include ${PLATFORM_PATH}/nativepower/nativepower.mk
include ${PLATFORM_PATH}/libcutils/libcutils.mk
......
```

4.3.4.3 加入 buildroot 编译配置中

进入 buildroot-201611 目录，运行 make menuconfig 目录，将 “boot-play package” 选中即可，如下图。

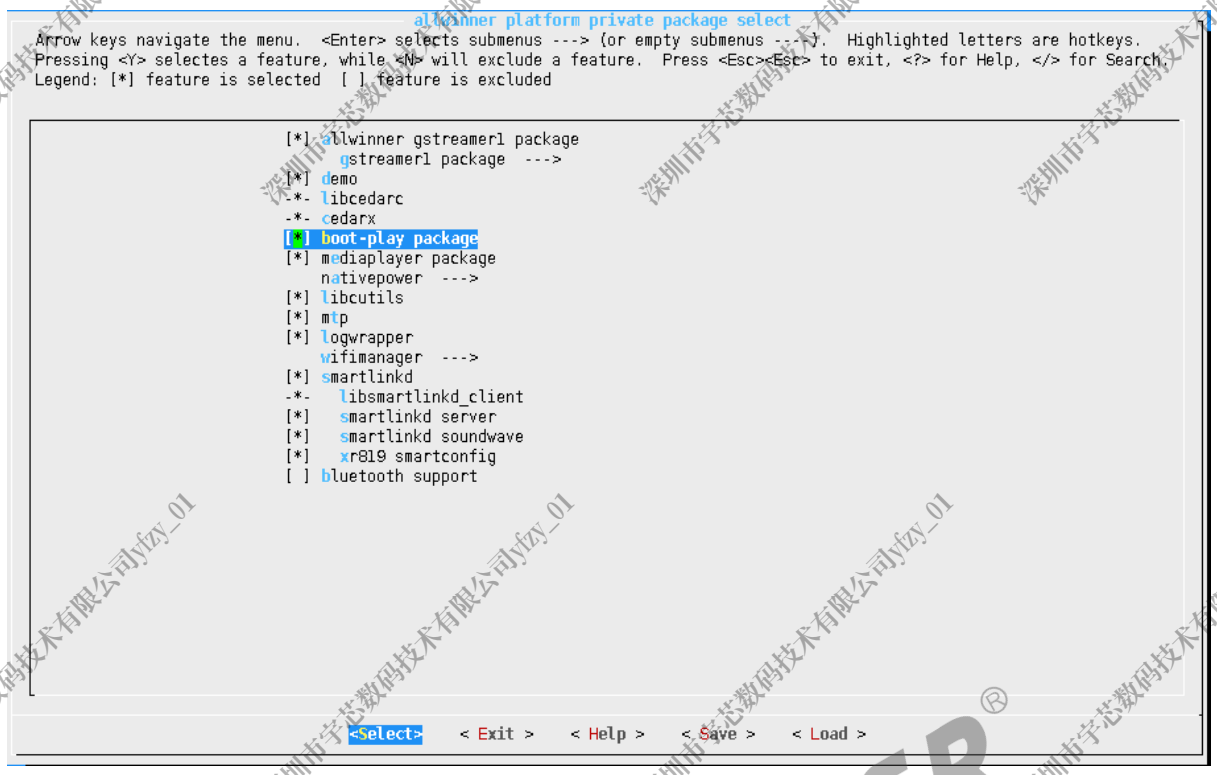

图 4-1: boot-play package

4.3.4.4 添加开机启动脚本和外部资源文件

longan 有三种方法将开机启动脚本添加到 rootfs 中，如下：

方法	说明	优缺点
Buildroot packge 编译机制	Buildroot 自带编译机制，在 packge.mk 中实现，编译 package 的同时也将开机启动脚本拷贝到 rootfs 中	优点：便于 buildroot 方案之间 package 移植缺点：非 packge 的启动脚本，无法拷贝到 rootfs 中
Buildroot post_build.sh 脚本	Buildroot 自带编译机制，在所有 package 编译完成，生成 rootfs.img 之前，将开机启动脚本拷贝到 rootfs 中	优点：1. 便于 buildroot 方案之间 package 移植一般拷贝非 packge 的启动脚本
longan COPY_FILES 机制	独立于 Buildroot 编译系统之外，在 buildroot 完整编译之后，将开机启动脚本拷贝到 rootfs 中，再生成 rootfs.img。	优点：1. 不依赖于构建平台，buildroot、yocto 都适用 2. 为了方便移植，建议拷贝与构建平台无关的资源文件。

修改 platform、config/buildroot/boot-play/boot-play.mk 脚本，如下：

```
.....
define BOOT_PLAY_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/initplay $(TARGET_DIR)/usr/bin
$(INSTALL) -D -m 0755 $(PLATFORM_PATH)/boot-play/S39initplay $(TARGET_DIR)/etc/init.d
//拷贝开机启动脚本到etc目录
endef
.....
```

添加外部资源文件, 修改 device/r328/perf1/perf1.mk，如下：

```
.....
#boot-play
PRODUCT_COPY_FILES += \
    device/common/base-files/boot-play:/usr/res/boot-play \
    device/common/base-files/etc/init.d/S39initplay:etc/init.d/S39initplay
.....
```

4.4 新增 ramfs 新功能

4.4.1 如何生成 ramfs

以 r328 为例，ramfs 涉及的文件有：

- buildroot/buildroot-201611/configs/sun8iw18p1_ramfs_defconfig：ramfs 配置文件
- platform/config/buildroot/busybox/init：ramfs 启动脚本
- platform/config/buildroot/post_init.sh：编译后处理脚本，脚本会在 ramfs 编译完成后，打包之前执行 |
- out/sun8iw18p1/std/buildroot/ramfs：amfs 编译输出目录 |
- device/config/chips/sun8iw18p1/configs/std/buildroot/BoardConfig.mk ： 脚 本 中 的 LICHEE_BR_RAMFS_DEF 表示 ramfs 默认配置文件

4.4.2 在 ramfs 中添加新功能

以在 ramfs 中增加 OTA 升级功能为例

- 步骤 1：将 ota-upgreade 加入 ramfs，执行./build.sh ramfs_menuconfig 进入配置界面。

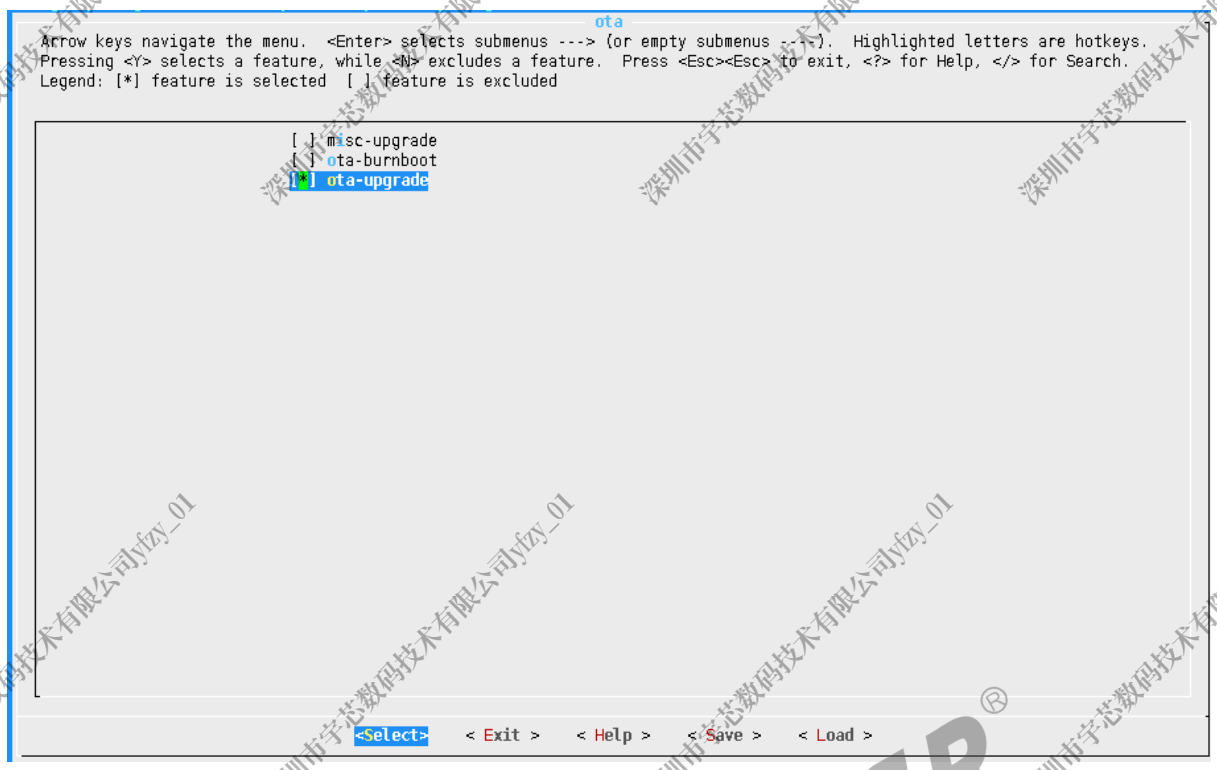

图 4-2: ota-upgreade

- 步骤 2：将 root_update 加入启动项

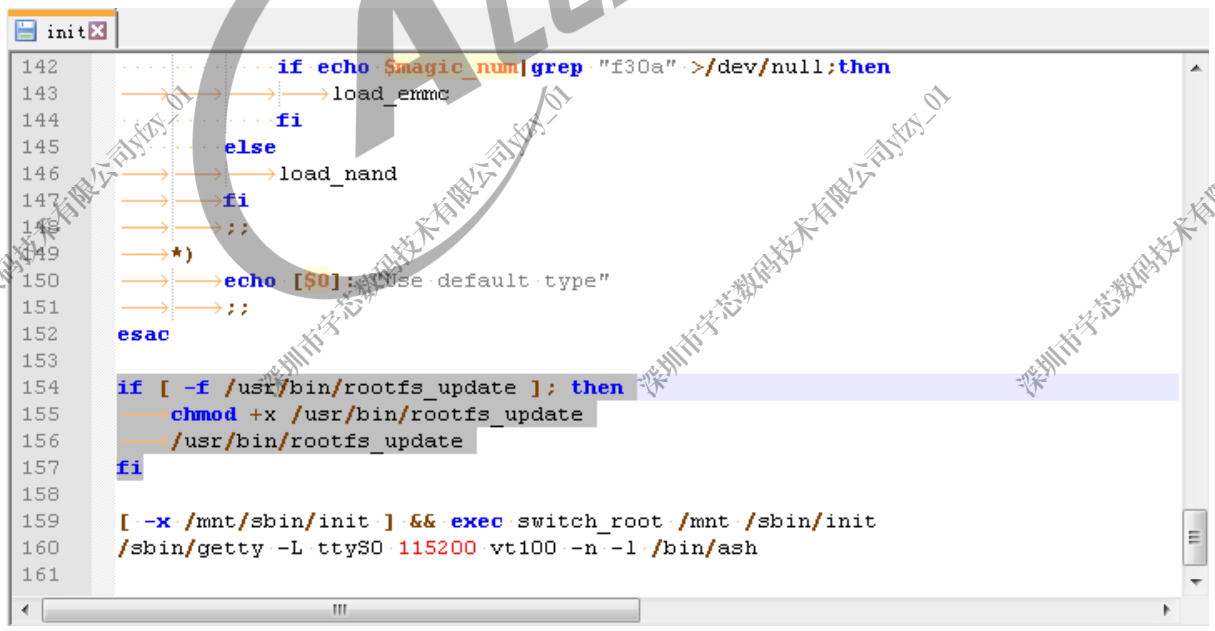

图 4-3: root_update

- 步骤 3：编译 ramfs，具体参考前面《Buildroot 配置和编译》中的 ramfs 编译方法。

5 其他说明

5.1 更换交叉编译工具链

5.1.1 kernel 更换交叉编译工具链

方法一 1：直接修改环境变量：

- 步骤 1：拷贝交叉编译工具链到 build/toolchain 目录中。
- 步骤 2：修改编译配置，修改 device/config 目录下 BoardConfig.mk 将 LICHEE_COMPILER_TAR 修改为需要更换的工具链名字。

方法 2：修改通用编译规则：

- 直接修改 build 目录下的 mkcmd.sh，打开该文件，找到下面的代码：

```
cross_compiler=(
'linux-3.4 arm   gcc-linaro.tar.bz2  target_arm.tar.bz2' #kernel arm kernel-gcc rootfs-gcc
'linux-3.4 arm64  gcc-linaro-aarch64.tar.xz  target_arm64.tar.bz2'
'linux-3.10 arm   gcc-linaro-arm.tar.xz  target_arm.tar.bz2'
'linux-3.10 arm64  gcc-linaro-aarch64.tar.xz  target_arm64.tar.bz2'
'linux-4.4 arm   gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz  target-arm-linaro-5.3.tar.bz2'
'linux-4.4 arm64  gcc-linaro-5.3.1-2016.05-x86_64_aarch64-linux-gnu.tar.xz  target-arm64-linaro-5.3.tar.bz2'
'linux-4.9 arm   gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz  target-arm-linaro-5.3.tar.bz2'
'linux-4.9 arm64  gcc-linaro-5.3.1-2016.05-x86_64_aarch64-linux-gnu.tar.xz  target-arm64-linaro-5.3.tar.bz2'
)
```

- 可以直接修改里面交叉编译链的名称。当然，该优先级要比修改 BoardConfig.mk 的低。

5.1.2 buildroot 更换交叉编译工具链

以 Buildroot-201611 支持 gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz 为例。

- 步骤 1：修改 Buildroot-201611/toolchain/toolchain-external/Config.in，增加配置选项。

```
diff --git a/toolchain/toolchain-external/Config.in b/toolchain/toolchain-external/Config.in
index 295d6bf..a82cd51 100644 (file)
--- a/toolchain/toolchain-external/Config.in
+++ b/toolchain/toolchain-external/Config.in
@@ -59,6 +59,29 @@ config BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMHF
                  with the hard floating point calling convention, and uses
                  the VFPv3-D16 FPU instructions.

+config BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMHF_2017
+             bool "Linaro ARM 2017.05"
+             depends on BR2_arm
+             depends on BR2_ARM_CPU_ARMV7A
+             depends on BR2_HOSTARCH = "x86_64" || BR2_HOSTARCH = "x86"
+             depends on BR2_ARM_EABIHF
+             depends on !BR2_STATIC_LIBS
+             select BR2_TOOLCHAIN_EXTERNAL_GLIBC
+             select BR2_TOOLCHAIN_HAS_SSP
+             select BR2_TOOLCHAIN_HAS_NATIVE_RPC
+             select BR2_INSTALL_LIBSTDCPP
+             select BR2_TOOLCHAIN_HEADERS_AT_LEAST_4_6
+             select BR2_TOOLCHAIN_GCC_AT_LEAST_6
+             select BR2_TOOLCHAIN_HAS_FORTRAN
+             help
+               Linaro toolchain for the ARM architecture. It uses Linaro
+               GCC 2016.05 (based on gcc 5.3.1), Linaro GDB 2016.05 (based on
+               GDB 7.11.1), glibc 2.21, Binutils 2016.05 (based on 2.25). It
+               generates code that runs on all Cortex-A profile devices,
+               but tuned for the Cortex-A9. The code generated is Thumb 2,
+               with the hard floating point calling convention, and uses
+               the VFPv3-D16 FPU instructions.
+
  config BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMEB
                bool "Linaro armeb 2016.05"
                depends on BR2_armeb
@@ -648,6 +671,7 @@ config BR2_TOOLCHAIN_EXTERNAL_PREFIX
                default "arceb-linux"                    if BR2_TOOLCHAIN_EXTERNAL_SYNOPSYS_ARC && BR2_arceb
                default "arm-linux-gnueabi"              if BR2_TOOLCHAIN_EXTERNAL_LINARO_ARM
                default "arm-linux-gnueabihf"            if BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMHF
+              default "arm-linux-gnueabihf"            if BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMHF_2017
                default "armeb-linux-gnueabihf"          if BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMEB
                default "arm-none-linux-gnueabi"         if BR2_TOOLCHAIN_EXTERNAL_CODESOURCERY_ARM
                default "arm-arago-linux-gnueabi"        if BR2_TOOLCHAIN_EXTERNAL_ARAGO_ARMV7A
```

图 5-1: Config.in

- 步骤 2：修改 Buildroot-201611/toolchain/toolchain-external/toolchain-external.mk，增加编译选项。

```
diff --git a/toolchain/toolchain-external/toolchain-external.mk b/toolchain/toolchain-external/toolchain-external.mk
index 9b3f0a2..3d947e2 100644 (file)
--- a/toolchain/toolchain-external/toolchain-external.mk
+++ b/toolchain/toolchain-external/toolchain-external.mk
@@ -334,6 +334,13 @@ TOOLCHAIN_EXTERNAL_SOURCE = gcc-linaro-5.3.1-2016.05-i686_arm-linux-gnueabihf.ta
  else
  TOOLCHAIN_EXTERNAL_SOURCE = gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabihf.tar.xz
  endif
+else ifeq ($(BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMHF_2017),y)
+TOOLCHAIN_EXTERNAL_SITE = https://releases.linaro.org/components/toolchain/binaries/6.4-2017.11/arm-linux-gnueabihf
+ifeq ($(HOSTARCH),x86)
+TOOLCHAIN_EXTERNAL_SOURCE = gcc-linaro-5.3.1-2016.05-i686_arm-linux-gnueabihf.tar.xz
+else
+TOOLCHAIN_EXTERNAL_SOURCE = gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz
+endif
  else ifeq ($(BR2_TOOLCHAIN_EXTERNAL_LINARO_ARMEB),y)
  TOOLCHAIN_EXTERNAL_SITE = https://releases.linaro.org/components/toolchain/binaries/5.3-2016.05/armeb-linux-gnueabihf
  ifeq ($(HOSTARCH),x86)
```

图 5-2: toolchain-external.mk

- 步骤 3：修改 Buildroot-201611/toolchain/toolchain-external/toolchain-external.hash，增加校验信息。

```
diff --git a/toolchain/toolchain-external/toolchain-external.hash b/toolchain/toolchain-external/toolchain-external.hash
index 85bc927..33f31bc 100644 (file)
--- a/toolchain/toolchain-external/toolchain-external.hash
+++ b/toolchain/toolchain-external/toolchain-external.hash
@@ -39,6 +39,7 @@ sha256  7f52ffcf490f489821a6ab6f73f7780b54d2a39361ddf1d709f12419aae4a8a1    gcc-lin
   sha256  77cccd433d680251c8b9bfa01890cb6921548513a39fac35c1f4b9f0a47e5aa4    gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabi.tar.xz
   sha256  f1421c580ce977226f4fefc9c409b3b423260cc65a6e9dc6da88bb3478a521a0    gcc-linaro-5.3.1-2016.05-i686_arm-linux-gnueabihf.tar.xz
   sha256  987941c9fffdf56ffcbe90e8984673c16648c477b537fcf43add22fa62f161cd    gcc-linaro-5.3.1-2016.05-x86_64_arm-linux-gnueabihf.tar.xz
+sha256  f52dbcce24da8b51e3bc65e26b3f72006707544bcf272ce014d3ff4313acf5ad    gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz
   sha256  f6e9c0d3320760fe8f89e9ec3acdd7b4da7eff889c094b4a2acc286fd46f334f    gcc-linaro-5.3.1-2016.05-i686_armeb-linux-gnueabihf.tar.xz
   sha256  6cf41c8944be56279cc14992aa075174b7a4c5938502536266eaaeef048f9440    gcc-linaro-5.3.1-2016.05-x86_64_armeb-linux-gnueabihf.tar.xz
   sha256  d43227248f282a652da42322fcf4abfd3021f2a2f62e0cf6e242d82f55966ba9    gcc-linaro-5.3.1-2016.05-i686_aarch64-linux-gnu.tar.xz
```

图 5-3: toolchain-external.hash

- 步骤 4：拷贝 toolschain 程序包到 Buildroot-201611/目录。

5.2 生成安全固件

5.2.1 修改配置

- 修改内存大小。对于 R328 来说，还需要依据 secure os 要使用的内存来修改内核保留内存，具体修改文件 linux-4.9/arch/arm/boot/dts/sun8iw18p1.dtsi。如果 secure os 使用了 4M 内存，可修改如下：

```
diff --git a/arch/arm/boot/dts/sun8iw18p1.dtsi b/arch/arm/boot/dts/sun8iw18p1.dtsi
index 589466f..90c4131 100644
--- a/arch/arm/boot/dts/sun8iw18p1.dtsi
+++ b/arch/arm/boot/dts/sun8iw18p1.dtsi
@@ -5,7 +5,7 @@
 */
 /* optee used */
-/memreserve/ 0x41a00000 0x00100000; /* optee range : [0x41a00000~0x41b00000], size = 1M */
+/memreserve/ 0x41900000 0x00400000; /* optee range : [0x41900000~0x41D00000], size = 4M */
 #include <dt-bindings/interrupt-controller/arm-gic.h>
 #include <dt-bindings/gpio/gpio.h>
```

- 配置防回滚版本号。芯片在引导固件的时候，会对比固件的版本号与芯片内存保留的版本号防回滚版本号在 device/config/chips/r328/configs/default/version_base.mk 中进行配置，文件中主要有两个属性可配置。

```
# define the verions of the image
# format: main
# such as 1, 2
# NOTICE: the range of main version is from 0 to 31,
# ROOT_ROLLBACK_USE ---0:not used,1:used
# when you change the version, you must increase main version, and never reduce the versions.
# the default version is 0

ROOT_ROLLBACK_USED = 1
MAIN_VERSION = 0
```

其中，ROOT_ROLLBACK_USED 表示是否使能配置，1 为使能，此时 MAIN_VERSION 的值会用作安全固件的防回滚版本号。0 为不加入，此时无论 MAIN_VERSION 的值为多少，安全固件的防回滚版本号固定为 0。未配置此项时，按 ROOT_ROLLBACK_USED = 0 处理。

MAIN_VERSION 表示固件的防回滚版本号，可用范围为 0-31。配置其他值芯片会直接认为固件版本号检验失败。

5.2.2 生成秘钥

运行 build/createkeys 工具，选择对应平台后，即可自动生成密钥对。

```
user@Exdroid47:~/workspace/r328/longan$ cd build
user@Exdroid47:~/workspace/r328/longan/build$ ./createkeys
All valid Sunxi chip:
0. sun8iw18p1
Please select a chip[sun8iw18p1]: 0
ready to create keys
/home/user/workspace/r328/longan/device/config/chips/sun8iw18p1
INFO: SELECT_CHIP is sun8iw18p1

"/home/user/workspace/r328/longan/device/config/chips/sun8iw18p1/configs/default/dragon_toc.cfg" -> "/home/user
      /workspace/r328/longan/device/config/dragon_toc.cfg"
keypath=/home/user/workspace/r328/longan/device/config/common/keys
create for Trustkey
Generating RSA private key, 2048 bit long modulus
e is 65537 (0x10001)
writing RSA key

.......

INFO: use platform[sun8iw18p1] toc config to creat keys
```

生成的 keys 文件在 out/$(IC)/common/keys 目录下。

5.2.3 生成安全固件

打包固件之前请确认 keys 文件在 out/$(IC)/common/keys 目录下。

生成安全固件方法：./build.sh pack_secure

5.2.4 生成安全固件的步骤

步骤 1：修改配置。

步骤 2：生成密钥，已经生成的，可以省略此步骤。将此密钥保存好，以免覆盖或丢失。

步骤 3：确认 keys 文件在 device/config/common/keys 目录下。

步骤 4：打包生成安全固件方法。

5.2.5 增加和删除开机启动服务

Rootfs 中/etc/init.d 目录是开机启动的相关的脚本。

- 增加和删减开机启动服务

增加和删除 S* 开头的文件，即可达到增加和删除开机启动服务的目的。

- 增加开机服务

方法 1: 编译 buildroot 软件包的同时，增加开机服务。

```
define SYSKLOGD_INSTALL_INIT_SYSV
    $(INSTALL) -m 755 -D package/sysklogd/S01logging \
        $(TARGET_DIR)/etc/init.d/S01logging
endef
```

图 5-4: Config.in

方法 2: 通过 buildroot 后处理脚本来实现，修改 platform/config/buildroot/post_build.sh。

```
add_preinit_to_inittab

cp ../../../buildroot/buildroot-201611/package/sysklogd/S01logging $(TARGET_DIR)/etc/init.d/S01logging
chmod 755 $(TARGET_DIR)/etc/init.d/S01logging
```

图 5-5: Config.in

方法 3: 通过 longan 的 copy_files 机制来实现，修改 device//target/drum/std/copy_files.mk

```
#sysklogd
PRODUCT_COPY_FILES += \
    buildroot/buildroot-201611/package/sysklogd/S01logging:etc/init.d/S01logging
```

图 5-6: Config.in

5.2.6 删除开机服务

通过 buildroot 后处理脚本来实现，修改 platform/config/buildroot/post_build.sh。

5.2.7 调整开机启动顺序

通过修改 S* 开头的文件名中的数字大小，来调整开机顺序，其中数字越小则文件的优先级越高。

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明





（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。